University of New Mexico
Department of Computer Science

Midterm Examination

CS 361 Data Structures and Algorithms
Spring, 2003

Name:
Email:

- Print your name and email, *neatly* in the space provided above; print your name at the upper right corner of *every* page. Please print legibly.
- This is an *closed book* exam. You are permitted to use *only* two pages of "cheat sheets" that you have brought to the exam. *Nothing else is permitted.*
- Do all four problems in this booklet. *Show your work!* You will not get partial credit if we cannot figure out how you arrived at your answer.
- Write your answers in the space provided for the corresponding problem. Let us know if you need more paper.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
- If any question is unclear, ask us for clarification.

Question	Points	Score	Grader
1	40		
2	20		
3	20		
4	20		
Total	100		

1. **Short Answer (40 points)**

True or False: (circle one, 4 points each)

(a) **True or False**: In a max-heap, the element with smallest key is always at the rightmost leaf node of the heap? *Solution: F: it's always at a leaf node but not necessarily the rightmost leaf node*

(b) **True or False**: The height of a heap on n nodes is $\Omega(\log n)$? *Solution: T: it's* $\Theta(\log n)$

(c) **True or False**: In a max-heap, the element with the largest key is always at the root of the heap? *Solution: T*

(d) **True or False**: Mergesort is asymptotically faster than heapsort (i.e. the big-O runtime of heapsort is better than the big-O runtime of mergesort)?*Solution: F: They are both* $O(n \log n)$ *time*

(e) **True or False**: Heapsort requires $O(n)$ extra space (not counting the space to store the array to be sorted)? *Solution: F: It requires* $O(1)$ *extra space, it's an in-place sorting algorithm*

Short Answer: For each function below, give a $\Theta()$ expression that is as simplified as possible. Justify your answers briefly. **Circle your final answer.** (4 points each)

(a) $n^2 \log n - n\sqrt{n} + 10 \log^{10} n$ *Solution:* $\Theta(n^2 \log n)$ *since this is the fastest growing term*

(b) $\log^2 n^5 + 10 \log n^{10}$ *Solution:* $\Theta(\log^2 n)$, *since* $\log^2 n^5 = 25 \log^2 n$, *and this is the fastest growing term*

(c) $\sqrt{n} \log^3 n + \log^4 n$ *Solution:* $\Theta(\sqrt{n} \log^3 n)$ *since* $\sqrt{n}$ *grows faster than* $\log n$

(d) $n * \sum_{i=0}^{n} 5^{-i}$ *Solution:* $\Theta(n)$ *since* $\sum_{i=0}^{n} 5^{-i} = O(1)$

(e) $2^{\log_4 n}$ *Solution:* $\Theta(\sqrt{n})$ *since* $\log_4 n = \log_2 n / \log_2 4 = \log_2 n/2$, *so* $2^{\log_4 n} = \Theta(\sqrt{n})$

1. **Short Answer (40 points), continued.**

2. **Annihilators and Recurrence Trees (20 points)**
Consider the recurrence: $T(n) = 3T(n/3) + n^2$ (and $T(n) = \Theta(1)$ for n a constant)

   (a) Use the recurrence tree method to get a "guess" (i.e. simplest possible big-O) on the solution to this recurrence. **You need not prove your guess correct.**

   (b) Now use annihilators (and change of variable) to get a tight upperbound (i.e. simplest possible big-O) on the solution to this recurrence.

Solution: **Recurrence Tree:** $T(n) = 3T(n/3) + n^2$, $T(n/3) = 3T(n/9) + (n/3)^2$, $T(n/9) = 3T(n/27) + (n/9)^2$. *Writing this out in a recurrence tree, we get that the zero level is one* n^2*, the first level is three* $n^2/9$*'s, the second level is* 9 $n^2/81$*'s. In general, the* i*-th level sums to* $n^2/3^i$*. There are* $\log_3 n$ *levels, so the sum of all of them is*

$$n^2 \sum_{i=0}^{\log_3 n - 1} 1/3^i \leq n^2 \sum_{i=0}^{\textit{infinity}} 1/3^i \quad (1)$$

$$= n^2 * (3/2) \quad (2)$$

Thus the solution to the recurrence is $O(n^2)$
Annihilators: *Let* $n = 3^i$ *and* $t(i) = T(3^i)$*. Then*

$$t(i) = 3t(i-1) + 3^{2i} \quad (3)$$

$$t(i) = 3t(i-1) + 9^i \quad (4)$$

The annihilator for this is $(\boldsymbol{L} - 3)(\boldsymbol{L} - 9)$*, and thus from the lookup table, the form of the recurrence is:*

$$t(i) = c_1 3^i + c_2 9^i \quad (5)$$

$$t(i) = c_1 3^i + c_2 (3^i)^2 \quad (6)$$

The reverse transformation gives that

$$T(n) = c_1 n + c_2 n^2$$

This is $O(n^2)$

2. **Annihilators and Recurrence Trees (20 points), continued.**

3. **Recursion and Recurrences (20 points)**

Consider the following function:

```
int f(int n){
  if (n==0) return 0;
  else if (n==1) return 1;
  else{
    int val = 3*f(n-1);
    val -= f(n-2);
    val -= f(n-2);
    return val;
  }
}
```

(a) Let $f(n)$ be the *value* returned by the function f when given input n. Write a recurrence relation for $f(n)$:

Solution: f(n) = 3f(n-1)-2f(n-2)

(b) Now solve the recurrence for $f(n)$ *exactly* using annihilators. (don't forget to check your solution)

Solution: Let $T_n = f(n)$, *and* $T = \langle T_n \rangle$. *Then*

$$T = \langle T_n \rangle \quad (7)$$
$$\boldsymbol{L}T = \langle T_{n+1} \rangle \quad (8)$$
$$\boldsymbol{L}^2 T = \langle T_{n+2} \rangle \quad (9)$$

Since $\langle T_{n+2} \rangle = \langle 3T_{n+1} - 2T_n \rangle$, *we know that* $\boldsymbol{L}^2 T - 3\boldsymbol{L}T + 2T = \langle 0 \rangle$, *and thus* $\boldsymbol{L}^2 - 3\boldsymbol{L} + 2 = (\boldsymbol{L} - 2)(\boldsymbol{L} - 1)$ *annihilates* T. *Thus* $f(n)$ *is of the form:*

$$f(n) = c_1 2^n + c_2 1^n$$

We know:

$$f(0) = 0 = c_1 + c_2 \quad (10)$$
$$f(1) \quad 1 = 2 * c_1 + c_2 \quad (11)$$

so $c_1 = 1$, $c_2 = -1$ *and thus*

$$f(n) = 2^n - 1$$

Check: $f(2) = 3$ *and* $2^2 - 1 = 3$.

3. **Recursion and Recurrences (20 points), continued.**

(c) Now let $T(n)$ be the *running time* of the algorithm f on the previous page when given input n. Write a recurrence relation for $T(n)$:
Solution: T(n) = T(n-1)+2T(n-2)+k for some constant k

(d) Now get a tight upperbound (i.e. big-O) on the solution for $T(n)$ using annihilators.
Solution: $\boldsymbol{L}^2 - \boldsymbol{L} - 2$ *annihilates the homogeneous part (factoring this gives* $(\boldsymbol{L}-2)(\boldsymbol{L}+1)$. $\boldsymbol{L}-1$ *annihilates the homogeneous part. So the total annihilator is* $(\boldsymbol{L}-2)(\boldsymbol{L}+1)(\boldsymbol{L}-1)$. *The lookup table tells us that*

$$T(n) = c_1 2^n + c_2(-1)^n + c_3 1^n$$

So the upperbound on the solution is $O(2^n)$

4. **Loop Invariants (20 points)**

In this question, you will be proving the correctness of the procedure *Heap-Increase-Key* using loop invariants. Recall that this procedure takes a heap A as input, and increases the key of the i-th node of A to the value "key". The procedure then ensures that the *max-heap property*(i.e. for all nodes j such that $1 < j \leq$heapsize (A), A[parent (j)] $\geq$ A[j]) is true for the new heap. The procedure is given below:

Heap-Increase-Key (A,i,key)

(a) if (key < A[i]) then error "new key is smaller than current key"

(b) A[i] = key;

(c) while (i>1 and A[Parent(i)] < A[i])

  i. do exchange A[i] and A[Parent(i)]

  ii. i = Parent(i);

Argue the correctness of *Heap-Increase-Key* using the following loop invariant:

At the start of each iteration of the while loop, the array A[1..heap-size(A)] satisfies the max-heap property , except that there may be one violation: A[i] may be larger than A[Parent(i)]

Show initialization,maintenance and termination for this loop invariant. (For termination, show that the max-heap property holds for A[1..heap-size(A)] after the while loop terminates)

Solution: **Initialization:** *At the start of the first iteration of the while loop, we have only changed the value of A[i] in the original heap. Thus A[1..heap-size(A)] satisfies the max-heap property except for the fact that A[i] may now be larger than A[Parent(i)]*

Maintenance: *Let i' be the value of i at the current iteration of the while loop. Now at the beginning of the current iteration of the while loop, the heap property holds for all A[1..heap-size(A)] except that A[i'] is larger than A[Parent(i')]. However, A[i'] and A[Parent(i')] are swapped during the current iteration. Thus the only new possible violation at the end of the loop iteration is that A[Parent(i')] is larger than A[Parent(Parent(i'))]. Setting i equal to Parent(i') at the end of the loop body then reestablishes the invariant.*

Termination: *We know that at the beginning of the last iteration of the while loop, the max-heap property held for A[1..heap-size(A)] except that there could be one violation: A[i] could be larger than A[Parent(i)] However, the while loop terminates only if* $i = 1$ *(i.e. there is no Parent(i)) or A[Parent(i)]* $\geq$ *A[i]. Thus, it's not the case that A[i] is larger than A[Parent(i)], and so the max-heap property holds with no violations!*

4. **Loop Invariants (20 points), continued.**